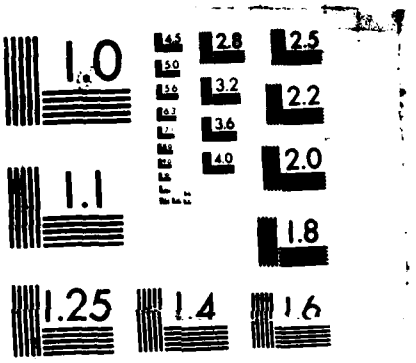
AD-A179 845 USING TRACE SPECIFICATIONS FOR PROGRAM SEMANTICS AND VERIFICATION(U) NAVAL RESEARCH LAB WASHINGTON DC J MCLEAN 10 APR 87 NRL-9033 1/1

UNCLASSIFIED F/G 12/5 NL



NRL Report 9033

Using Trace Specifications for Program Semantics and Verification

JOHN MCLEAN

Computer Science and Systems Branch
Information Technology Division

April 10, 1987

AD-A179 845

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NRL Report 9033	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Research Laboratory 6b. OFFICE SYMBOL (If applicable) Code 5590	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Washington, DC 20375-5000	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Office of Naval Research 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 61153N; PROJECT NO.; TASK NO R8014-09-41; WORK UNIT ACCESSION NO. DN480-540

11. TITLE (Include Security Classification)
Using Trace Specifications for Program Semantics and Verification

12. PERSONAL AUTHOR(S)
McLean, John

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Interim	FROM 10/85 TO 9/86	1987 April 10	17

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Software verification; Program semantics; Software specification

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Traditional methods for proving program correctness use implementation-dependent specification methods. If abstract specifications are also used, these methods require a leap of faith to bridge the gap between an abstract specification and a program correctness statement. In this report the trace method of software specification is extended to provide a natural semantics for procedural programming languages. This extension is compared with other approaches for giving program semantics and is seen to provide a method for proving program correctness that avoids the problems of those currently in use.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL John McLean	22b TELEPHONE (Include Area Code) (202) 767-3381 22c OFFICE SYMBOL Code 5590

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

GPO: 1986—607-047

CONTENTS

USING TRACE SPECIFICATIONS FOR PROGRAM SEMANTICS AND VERIFICATION

1. INTRODUCTION

Traditionally, program verification has focused on proving isolated programs and their subroutines correct *vis-à-vis* their specifications. Unfortunately, working with isolated programs has resulted in faulty program specification [1]. The problem with specifying programs individually is that many programs do not have any user-visible effects except for their interaction with other programs. For example, the effect of executing a push procedure in a stack module can be seen only by its interaction with other procedures that permit a user to examine the stack. To specify such programs, verifiers have hitherto been forced to tell programmers too little about what to do and too much about how to do it. They have resorted to pseudocode, algorithms, hidden functions, program states, and user-invisible variables to describe program interaction. This leads to a loss of specification abstraction that forces the module programmer, the module user, and later the system maintainer to glean essentials from a mass of implementation clutter [2]. Often, they disagree on what is essential and what is notational artifact. The result is often unnecessary module coupling that makes software extremely difficult to maintain.

To alleviate this problem, some verification methods supplement implementation-dependent specifications designed for the verifier, with abstract specifications designed for the programmer, system user, and system maintainer. However, there often has been a leap of faith required to get from the correctness of the former specification to the correctness of the latter. What is required is a coherent framework that allows for both the abstract specification of interacting modules and correctness assertions for these modules.

The trace method for specifying software [2,3] is a formal, abstract specification language that allows the specifier to describe a program's behavior in terms of other programs with which it interacts. In Ref. 2 a formal foundation for the method is presented with several examples of how to apply the method. This report continues the work in Ref. 2 by showing how the method can be extended to provide a natural semantics for procedural programming languages. This semantics provides a coherent framework for proving programs correct that avoids the problems inherent in the traditional approach.

2. AN OVERVIEW OF THE EXTENDED TRACE LANGUAGE

We begin by extending the trace specification language so that it can describe the effects of executing single program statements. In Ref. 2 a trace specification for a module consists of two parts: (1) a syntax section that gives the procedure names and types the module comprises, and (2) a semantics section that gives the behavior that the module's procedures must exhibit. Procedure behavior is given by listing assertions that describe the behavior of sequences of procedure calls, written $call_1.call_2.\cdots.call_n$, known as *traces*. The assertions are based on first order logic, supplemented by the predicate L, which is true when applied to a *legal* trace (a trace that does not cause an error), and the function V, which gives a return value when applied to a legal trace ending in a function call. The null trace, denoted by [], is always legal and is such that for any trace T, $T=[].T=T.[]$. Two traces T_1 and T_2 are *equivalent*, written $T_1 \equiv T_2$, if and only if for any trace T, $L(T_1.T)$ *iff* $L(T_2.T)$ and for nonnull T, $V(T_1.T)=V(T_2.T)$ if defined. Less formally, if two traces are equivalent, then

Manuscript approved November 12, 1986.

they are indistinguishable as far as L and V are concerned with respect to future program behavior. As an example, consider the following specification for a simple stack:

STACK SPECIFICATION

Syntax:

PUSH: integer
POP: $\rightarrow$ integer

Semantics:

(1) $L(T) \rightarrow L(T.PUSH(n))$
(2) $L(T) \rightarrow T.PUSH(n).POP \equiv T$
(3) $L(T) \rightarrow V(T.PUSH(n).POP) = n$

The syntax section says that the procedure *PUSH* takes an integer as a parameter and that *POP* returns an integer. The first assertion in the semantics section says that if T is a legal sequence of procedure calls, then the sequence consisting of T followed by a call to the procedure *PUSH* with any integer as a parameter is legal. The second assertion says that if T is a legal sequence of procedure calls, then T is equivalent to the sequence of calls consisting of T followed by a call to *PUSH* and a call to *POP*. The third assertion, when coupled with the first two, says that *POP* returns the last value pushed on the stack that has not previously been popped. Note that *PUSH* and *POP* are completely and unambiguously specified without suggesting an implementation.

In Ref. 2 a model-theoretic semantics specifies what assertions are semantic consequences of a specification, a derivation system specifies what assertions are derivable from a specification, and completeness and soundness theorems show that an assertion is a semantic consequence of a specification if and only if it is derivable from the specification. Among other things, this supports coextensive semantic and syntactic definitions of *totalness* and *consistency* and formal techniques for proving these properties for a specification. For our purposes here, the most important elements of this foundation are the following axioms from the derivation system. The first two say that the empty trace is legal and that any initial segment of a legal trace is legal. The third is a definition of equivalence.

TRACE AXIOMS

(1) $L([])$

(2) $L(T.T_1) \rightarrow L(T)$

(3) $T_1 \equiv T_2 \longrightarrow$

$(T)((L(T_1.T) \longleftrightarrow L(T_2.T)) \wedge$

$(T \neq [] \rightarrow$

$((\exists x)V(T_1.T) = x \rightarrow V(T_1.T) = V(T_2.T))))$*

For proving program correctness we also find use for the following induction schema:

INDUCTION SCHEMA

For any property P, if $P([])$ and $P(\phi) \rightarrow (x_1) \cdots (x_n)P(\phi.C(x_1, \cdots, x_n))$ for each procedure call C that take n variables where $x_1, \cdots, x_n$ are not in P, then $P(T)$ where T is not in P.

*This is a simpler definition than given in Ref. 2, but the two definitions can easily be shown to be equivalent.

This schema is sound with respect to the semantics given in Ref. 2 if we limit the domain of traces to the null trace and sequences of procedure calls. This was not done in the original presentation where infinite traces were allowed for the sake of completeness. Our limitation renders the resulting system incomplete in the sense that although $\{P([]),P(C),P(C.C),\cdots\} \mid = P(T)$ in a specification that has the single parameterless procedure call C, we cannot derive $P(T)$ from the given assumption set. In general, it will no longer be true that we can derive every consequence of an infinite set of assumptions from that set, but we can still derive every consequence of a finite set of assumptions from that set. In technical terms, we have given up *compactness*, but this is a small price to pay for the ability to prove programs correct [4].*

We must also extend the notion of *trace* to include strings of program statements. This necessitates introducing program variables and operators into the language as primitives and introducing statement variables that are like trace variables except that they range over sequences of program statements instead of sequences of procedure calls. Since a sequence of procedure calls is a sequence of program statements, a sequence of procedure calls or procedure call variables is a valid substitution instance for a statement variable.† We use T, possibly subscripted, as a procedure call variable and S, possibly subscripted, as a statement variable. The rest of the language is unaffected except for the predicate V. Whereas in Ref. 2, V was a function from traces to value domains, in the extended language, V takes two arguments—a trace and a program variable—and returns the value of the program variable after the execution of the given trace. When the trace ends in a procedure call and the program variable is the return value of that call, the second argument can be omitted. For all intents, this renders the extended language a superset of the original language.‡

3. PROGRAM SEMANTICS

In this section we present a procedural programming language and show how traces can be used to give the semantics of the language.§ Since the trace language contains Boolean expressions and can easily be extended to contain other data types such as integers, lists, or arrays, we are not concerned with their semantics. For simplicity we limit our programming language to Booleans, integers, and arrays of integers, and we assume axioms for integer and Boolean operators. Our primary focus is on giving the semantics of the control structures of our language.

We limit ourselves to what Linger, Mills, and Witt call *proper programs* [5]. These are programs that, intuitively, (a) have a single entry point and a single exit point, and (b) for each line of code, have a control path through that line from the entry point to the exit point. The second condition rules out programs that contain code that is *syntactically* unreachable or unleavable. For example, it rules out programs containing code of the form

```
A: goto B;
   a:=a+1;
B: goto A;
```

*See Ref. 4 for a discussion of this problem with respect to temporal logic.

†But a sequence of program statements is not a valid substitution for a procedure call variable. The reason for the distinction is to restrict assertions, such as the induction schema above, to sequences of procedure calls.

‡The qualifier is necessary since in Ref. 2 V was defined only on legal traces ending in a function call, and in the extended system V can be defined for illegal traces as well. This is of small matter, however, since we can either alter Ref. 2 t fit by regarding V as being defined on an unspecified set of traces that includes its original domain or by treating V as being systematically ambiguous, letting context decide.

§For nonprocedural languages, such as Prolog [6], proving correctness is relatively easy except for language idiosyncrasies as described in Ref. 7.

since no evaluation must be made to determine that the code from A to B is unleavable and that the command $a:=a+1$ is unreachable. However, it does not rule out programs that contain code of the form

```
while (x=x)
  {if (x=x) then {a:=a+1} else {b:=b+1}}
```

since one can determine that the loop is unleavable and that $b:=b+1$ is unreachable only by examining the expressions contained in the program.

Excluding nonleavable code and nonreachable code from our programming language does not reduce expressive power since the former can be replaced by any nonterminating loop and the latter can be excised without altering program behavior. Nor does limiting ourselves to programs with a single entry and a single exit point reduce expressive power since we can always embed any program that violates this principle within a proper program. By limiting ourselves to proper programs, however, we have a real gain in light of the Structure Theorem [5]. This theorem states that any proper program can be written in a language whose only control structures besides simple sequencing are *WHILE DO* and *IF THEN ELSE*. This motivates the language *SIMPLE* defined below. As mentioned above, we assume the set VBL of integer program variables, BOOL of Boolean expressions, and EXPR of integer expressions. We are not interested in variable declarations.

SIMPLE

```
PROGRAM →
  PROCEDURE NAME[(VBL,...,VBL)]: [RETURN(VBL)] STATEMENT.

STATEMENT →
  skip |
  ASSIGNMENT |
  IF THEN ELSE |
  WHILE DO |
  STATEMENT; STATEMENT

ASSIGNMENT →
  VBL := EXPR
  VBL := PROCEDURE NAME[(VBL,...,VBL)]

IF THEN ELSE →
  if (BOOL) then {STATEMENT} else {STATEMENT}

WHILE DO →
  while (BOOL) {STATEMENT}
```

We assume without loss of generality that SIMPLE does not contain recursion since we can always eliminate it [8]. We also assume without loss of generality that all variables are global and that integer variables are initialized to 0.* Finally, we will abbreviate program statements of the form **if** (ϕ) **then** $\{\gamma\}$ **else** {**skip**} as **if** (ϕ) **then** $\{\gamma\}$.

*The effect of local variables can be achieved by judicious naming.

The semantics for SIMPLE is given by the following axioms (where parentheses and brackets are omitted around Greek letters for readability):

PROGRAM SEMANTIC AXIOMS

(1) V(S,c)=c, for constant c.
(2) V([],x)=0, for any integer variable x.
(3) V(S.a:=t,a)=V(S,t), for term t.
(4) V(S,a *op* b)= V(S,a) *op* V(S,b) for arithmetic operation *op*.
(5) V(S,a *op* b) ⟷ V(S,a) *op* V(S,b) for Boolean operation *op*.
(6) V(S.a:=x,b)=V(S,b) where b is independent, as defined below, of a.
(7) V(S,ϕ)=V(S,ψ) → V(S,a[ϕ])=V(S,a[ψ]) where a is an array.
(8) V(S,ϕ)≠V(S,ψ) → V(S.a[ϕ]:=t,a[ψ])=V(S,a[ψ]) where a is an array
(9) V(S,ϕ) → V(S.if ϕ then θ else ψ,x) = V(S.θ,x)
(10) ¬V(S,ϕ) → V(S.if ϕ then θ else ψ,x) = V(S.ψ,x)
(11) V(S.while ϕ do θ,x)=V(S.if ϕ then {θ.while ϕ do θ},x)
(12) V(S.skip,t)=V(S,t), for term t.

We say that a is *dependent* on b if (1) a is typographically identical to b or (2) if a is an array variable of the form $\alpha[\phi]$ and b is either dependent on ϕ or of the form $\alpha[\psi]$ for the same array α. That is, a is dependent on b if they are identical, if a is an array variable whose index depends on b, or if both are array variables to the same array. Note that in the second case, a is dependent on b, but not *vice versa*. We say that a is *independent* of b if a is not dependent on b. The intuition behind the definitions is to restrict axiom (6) so that altering $a[t]$ may alter $a[y]$ if $t=y$ and altering t may alter, e.g., $a[t]$ or $a[t+1]$.

Although the axioms are sufficient only for proving weak program correctness, i.e., that if the program terminates then it produces the correct answer, we can extend them to prove strong program correctness, i.e., that the program is weak correct and terminates, in a straightforward manner. Clearly, a nonrecursive SIMPLE program terminates if all of its loops terminate, so we can restrict ourselves to loop termination. Consider the relation $ACC(\phi,\psi,\theta)$, which intuitively says that trace θ can be formed from trace ϕ by appending zero or more occurrences of ψ. We can recursively define such a predicate by adding the following axiom to our system:

(13) ACC(R,S,R) ∧ (ACC(R,S,T) → ACC(R,S,T S))

Given this axiom, we can replace axiom (11) by the following axiom, which enables us to prove strong correctness:

(11′) (¬ V(S,ϕ) ∧ ACC(T,θ,S)) → V(T.while ϕ do θ,x)=V(T.if ϕ then {θ.while ϕ do θ},x)

We can add axioms for proving termination of recursive programs in a similar manner, but for simplicity, we shall limit ourselves to the original 12 axioms in the rest of this report.

The axioms are based on the standard Hoare axioms [9] for weak correctness. However, the semantic axioms given here differ from Hoare's in that they are stated in terms of variable values instead of in terms of general preconditions and postconditions. A Hoare-style assertion such as $A\{p\}B$ (if A is true before executing program p, then B is true after completion of p) can be stated in the present system as $A^T \rightarrow B^T$ where A^T is the same as A except that every term t is replaced by $V(S,t)$ for some statement variable S, and B^T is the same as B except that every term t is replaced by $V(S.p,t)$. Hence, $x=0\{x:=x+1;y:=x\}y=1$ is equivalent to $V(S,x)=0 \rightarrow V(S.x:=x+1.y:=x,y)=1$.

It might seem as though the correct translation should be $V([],x)=0 \rightarrow V(S.x:=x+1.y:=x,y)=1$. However, such a strategy fails when we consider the assertion $x=1\{x:=x+1;y:=x\}y=1$ since $V([],x)=1$ is false given our assumption that all uninitialized integers are 0. If there is no

sequence of programs that can set x to 0, then $V(S,x)=0$ will also fail to hold for all S. This may seem to be a disadvantage of the trace approach, but it fits well with our desire for abstraction. If a certain state is unrealizable by a module, we have no business placing restrictions on what would happen if that state were realized. Further, if we assume that our language contains a deterministic assignment statement such as epitomized by **x:=0**, then all possible states are realizable.

Since we can translate Hoare-style assertions into trace assertions, we can define any language construct that is definable by Hoare-style assertions. For example, consider Dijkstra's **if fi** and **do od** constructs [10]. The semantics for the former is given by the two axiom schemata $((V(S,b_1) \vee \cdots \vee V(S,b_n)) \wedge V(S,b_1) \rightarrow V(S.p_1,x)=\phi \wedge \cdots \wedge V(S,b_n) \rightarrow V(S.p_n,x)=\phi) \rightarrow V(S.if\, b_1 \rightarrow p_1 | \cdots | b_n \rightarrow p_n fi,x)=\phi$ and $\neg(V(S,b_1) \vee \cdots \vee V(S,b_n)) \rightarrow \neg(\exists y)V(S.if\, b_1 \rightarrow p_1 | \cdots | b_n \rightarrow p_n fi,x)=y$. The semantics for the latter is given by $V(S.do\ b_1 \rightarrow p_1 | \cdots | b_n \rightarrow p_n od,x)= V(S.if\,(b_1 \vee \cdots \vee b_n)$ then $\{$if $b_1 \rightarrow p_1 | \cdots | b_n \rightarrow p_n fi;\ do\ b_1 \rightarrow p_1 | \cdots | b_n \rightarrow p_n od\},x)$.

An advantage of the semantic axioms given here is that when using them for proving programs correct, we do not have to worry about finding proper initial conditions. Finding such conditions for Hoare-style axioms is a nontrivial task often neglected in the literature. A second advantage is that we allow trace variables. This gives us a more expressive language. For example, although our stack assertion (3) can be rendered as the Hoare-style assertion $true\{PUSH(n).POP\}return=n$, we cannot capture by any Hoare-style assertion assertion (2) of our specification. It may seem as though $P\{PUSH(n).POP\}P$, i.e., whatever is true before calling *PUSH* then calling *POP* is true after the calls, approximates our concept of *equivalence*, but this is not the case. It is not the case that anything that is true before $PUSH(n).POP$ is true after $PUSH(n).POP$, only that anything that is user visible that is true before the call is true after the call. For example, if our stack is implemented as an array, the array state will be different, but not in a way that a module user can detect. More on this in the next section.

Our language can also express assertions of dynamic logic [11] if we allow for nondeterminancy by letting V return a set of values [12], thus, treating V as a relation rather than as a function. The result is a formulation of dynamic logic that employs standard first order model theory instead of ω-sequences of models. For example, if we let $V(S,t,v)$ mean that the value of t can be v after executing S, then the $\cup$ construct, which has the property that $p \cup q$ executes either program p or q, can be specified by $V(p \cup q,t,x) \longleftrightarrow V(p,t,x) \vee V(q,t,x)$. Similarly, the * construct, which has the property that for any program p, p^* executes p 0 or more times, can be specified by the axiom schema $V(p^*,t,x_i)$ where x_i is the value of executing p i times. The statement $x=0 \rightarrow <p>x=1$ (i.e., if $x=0$ then $x=1$ is true in some state reachable by executing program p) is represented by $(v)(V(S,x,v) \rightarrow v=0) \rightarrow V(S.p,x,1)$, and $x=0 \rightarrow [p]x=1$ (i.e., if $x=0$ then $x=1$ is true in every state reachable by executing program p) is represented by $(v)(V(S,x,v) \rightarrow v=0) \rightarrow (v)(V(S.p,x,v) \rightarrow v=1)$.* Analogous comments apply here as were given when considering the Hoare-style approach. The translations are faithful if we consider the antecedent $x=0$ to apply only to states that are realizable by a sequence of procedure calls. Further, we must assume that this sequence always leads to a state in which $x=0$. As above, these restrictions are easy to satisfy if we assume that we have deterministic assignment. Some of the advantages of our approach over dynamic logic are analogous to the advantages of our approach to that of Hoare-style logic. A further advantage is that we avoid the complicated model theory that dynamic logic necessitates since we do not need to consider ω-sequences of program states.

4. PROGRAM CORRECTNESS

As an example of a program correctness proof, consider the following program for computing factorial:

*Of course, the relational counterpart of V can also be used to give semantics for **if fi** and **do od**.

FACTORIAL PROGRAM

```
FAC(n): RETURN(fac)
fac:=1;
i:=n;
while (i>0)
  {fac:=fac*i;
   i:=i-1}.
```

We prove FAC correct with respect to the simple specification $V(FAC(0))=1 \wedge (x \geq 0 \rightarrow V(FAC(x+1))=(x+1)*V(FAC(x)))$ by deriving the specification from FAC, our program semantic axioms, and the trace axioms of Ref. 2.

We consider each conjunct in turn. The specification then follows by simple sentential logic. To derive $V(FAC(0))=1$, note that by definition $V(FAC(0))=V(fac:\ \ =1.i:=0.\ while\ (i>0)\{fac:=fac*i;i:=i-1\},fac)$. By axiom (11), this is equal to $V(fac:=1.i:=0.if\ (i>0)$ then $\{fac:=fac*i;i:=i-1;\ while\ (i>0)\{fac:=fac*i;i:=i-1,\}\}fac)$. Since by axiom (1) $V(fac:=1.i:=0,i)=0$, we can use axiom (10) (and implicitly axiom (12)) to derive $V(FAC(0))=V(fac:=1.i:=0,fac)$. By axioms (6) and (1), $V(fac:=1.i:=0,fac)=1$, and we are done.

To derive $x \geq 0 \rightarrow V(FAC(x+1))=(x+1)*V(FAC(x))$ requires a lemma. We prove by induction on the integers that for any integer m, $(m=V(S_1,i) \wedge m=V(S_2,i) \wedge V(S_1,fac)=V(S_2,fac)*\gamma) \rightarrow V(S_1\ .while(i>0)\ \{fac:=fac*i;i:=i-1\},fac)=\ V(S_2.while(i>0)\ \{fac:\ \ =fac*i;i:=i-1\},fac)*\gamma$. It is trivial to prove this for $m=0$ so we assume it for $m=n \geq 0$ and show it for $m=n+1$. Note that if $m=n+1$ and $n \geq 0$, then by using axiom (11) the lemma is equivalent to $(n+1=V(S_1,i) \wedge n+1=V(S_2,i) \wedge V(S_1,fac)=V\ (S_2,fac)*\gamma) \rightarrow V(S_1.fac:=fac\ \ *i.i:=i-1.while\ (i>0)\ \ \{fac:=fac*i;i:=i-1\},fac)=\ \ V(S_2.fac:=fac\ \ *i.i:=i-1.while(i>0)\ \ \{fac:=fac*i;i:=i-1\},fac)*\gamma$. But $(n+1=V(S_1,i) \wedge n+1=V(S_2,i) \rightarrow n=V(S_1.fac:=fac\ \ *i.i:=i-1,i)=V\ (S_2.fac:=fac\ \ *i.i:=i-1,i)$. Further $(V(S_1,fac)=V\ (S_2,fac)*\gamma \wedge\ V(S_1,i)=V\ (S_2,i)) \rightarrow V(S_1.fac:=fac\ \ *i.i:=i-1,fac)=V\ (S_2.fac:=fac\ \ *i.i:=i-1,fac)*\gamma$. Hence, by the induction hypothesis, since S_1 and S_2 are statement variables, $V(S_1.fac:=fac\ \ *i.i:=i-1.while(i>0)\ \{fac:=fac*i;i:=i-1\},fac)\ =\ V(S_2.fac:=fac\ \ *i.i:=i-1.while(i>0)\ \{fac:=fac*i;i:=i-1\},fac)*\gamma$, and we are done with our lemma.

We can now derive $x \geq 0 \rightarrow V\ (FAC(x+1))=(x+1)\ \ *V(FAC(x))$. Note that $V(fac:=1.i:=x,i)=x$ and $V(fac:=1.i:=x+1.fac:=fac*i.i:=i-1,i)=x$. Also, $V(fac:=1.i:\ \ =x,fac)*(x+1)=V(fac:=1.i:=x+1.fac:=fac\ \ *i.i:=i-1,fac)$. Therefore, by our lemma, $V(fac:=1.i:=x.\ while(i>0)\ \{fac:=fac*i;i:=i-1\},fac)*(x+1)=\ V(fac:=1.i:=x+1.fac:=fac\ \ *i.i:=i-1.while\ (i>0)\{fac:=fac*i;i:=i-1\},fac)$. But $V(fac:=1.i:=x.while(i>0)\{fac:=fac*i;i:=i-1\},fac)=V(FAC(x))$, and $x \geq 0 \rightarrow\ V(fac:=1.i:=x+1.fac:=fac\ \ *i.i:=i-1.while(i>0)\ \{fac:=fac*i;i:=i-1\},fac)=\ V(FAC(x+1))$. Hence, $x \geq 0 \rightarrow V(FAC(x))\ \ *(x+1)=V\ (FAC(x+1))$, and we are done.

As a second example of a program correctness proof, consider the stack specification given in Section 2. In this section we show how to prove a SIMPLE implementation of the specification to be correct. We use the following program where **ptr** and **top** are integer variables and **stack** is an integer array.

STACK PROGRAM

```
PUSH(i):
if (ptr≥0) then {ptr := ptr+1; stack[ptr]:= i}.

POP: RETURN(top)
if (ptr>0) then {top:=stack[ptr]};
ptr:=ptr-1.
```

As before, a correctness proof consists of a derivation of the specification from the program using the the axioms of Ref. 2 and the axioms that define the semantics of SIMPLE. However, this is not as straightforward as the in factorial case. It is not clear how to treat some of the constructs found in our specification [1]. For example, although it is clear how to derive the assertion $V(PUSH(n).POP)=n$, it is not clear how to derive $L(T) \rightarrow V\ (T.PUSH(n).POP)=n$. We could side-step this problem by rewriting our program so that $V(T.PUSH(n).POP)=n$ is always true. However, such a "solution" may not always be available. Further, legality is not the only troublesome concept; the same problem arises for equivalence as well.

What is clearly needed are program-counterparts to legality and equivalence. Logically, this is equivalent to giving an interpretation for the two predicates, which is perfectly all right as long as the interpretation preserves the truth of the relevant axioms from the trace derivation system. Intuitively, a stack trace is legal if it does not try to pop an empty stack. Since popping an empty stack decrements the variable *ptr* below 0 and a negative *ptr* can never become nonnegative, we can say that a trace T is legal *iff* $V(T,ptr)\geq 0$. Two traces T_1 and T_2 are equivalent if they have the same value for *ptr* and the same value for the variables $stack[1], \cdots, stack[ptr]$. In other words T_1 is equivalent to T_2 *iff* $V(T_1,ptr)=V\ (T_2,ptr) \wedge (1\leq i\leq V\ (T_1,ptr) \rightarrow V\ (T_1,stack[i])=V(T_2,stack[i]))$. For correctness, we need to establish that the relevant legality and equivalence axioms of Ref. 2 hold for our interpretation and that we can derive the stack specification given our interpretation. In other words, we must show that we can derive the stack specification and our three trace axioms from our program semantic axioms, our stack implementation, and the two axioms $L(T) \longleftrightarrow V\ (T,ptr)\geq 0$ and $T_1 \equiv T_2 \longleftrightarrow (V(T_1,ptr)=V\ (T_2,ptr) \wedge (1\leq i\leq V\ (T_1,ptr) \rightarrow V\ (T_1,stack[i])=V\ (T_2,stack[i])))$.

It is up to the programmer to provide these program-relative definitions for *legality* and *equivalence*. The programmer should formulate them as the program is being written and use them to verify informally the code. The program verifier uses them to derive formally the specification. This is equivalent to using our implementation and program semantic axioms to derive the following assertions:

STACK CORRECTNESS ASSERTIONS

(1) $V(T,ptr)\geq 0 \rightarrow V(T.PUSH(n),ptr)\geq 0$

(2) $V(T,ptr)\geq 0 \rightarrow (V(T,ptr)=V(T.PUSH(n).POP,ptr) \wedge (1\leq i\leq V(T,ptr) \rightarrow V(T,stack[i])=V(T.PUSH(n).POP,stack[i])))$

(3) $V(T,ptr)\geq 0 \rightarrow V(T.PUSH(n).POP,top)=n$

(4) $V([],ptr)\geq 0$

(5) $V(T.T_1,ptr)\geq 0 \rightarrow V(T,ptr)\geq 0$

(6) $(V(T,ptr)=V(T_1,ptr) \wedge (1\leq i\leq V(T,ptr) \rightarrow V(T,stack[i])=V(T_1,stack[i]))) \longrightarrow (T_2)((V(T.T_2,ptr)\geq 0 \longleftrightarrow V(T_1.T_2,ptr)\geq 0) \wedge (T_2\neq [] \rightarrow ((Ev)V(T.T_2)=v \rightarrow V(T.T_2)=V(T_1.T_2))))$

The derivations of these assertions are included as an appendix to this report. Here, it is worthwhile to consider the assertions themselves. Assertions (1) to (3) correspond to assertions (1) to (3) of the stack specification, and assertions (4) to (6) correspond to trace axioms (1) to (3) given above. These assertions are "implementation-relative" versions of the corresponding specification assertions and trace axioms. This does not imply that the original implementation-free versions are functionless, however. The implementation-free specification provides a foundation for all implementations; it is what the programmer uses to understand what the module should do. Similarly the implementation-free trace axioms provide a foundation for all specifications. The programmer uses

both to formulate program-relative concepts of legality and equivalence. If we were to consider a different implementation, say one where the stack was implemented as a linked list, a variable-length character string, or a Gödel number, the correctness assertions might differ to reflect different program-relative definitions of *legality* and *equivalence*, but they would still be counterparts of the same specification assertions and trace axioms. The advantage of the trace approach is that it combines abstract specifications with program correctness statements in a coherent manner. The program specification leaves the programmer free to choose the best implementation, yet provides a framework for the programmer to formulate program correctness assertions from the program later.

Note that if we used a faulty counterpart for legality or equivalence, we would be unable to prove the counterparts of our trace axioms. For example, if we called a trace T legal if $V(T,ptr)>0$, then we would have been unable to prove that $L([])$. Hence, although some care may be required in finding the correct counterparts to equivalence and legality, our choice is subject to verification. Further, since all our derivations are formalizable in a rigorous deductive system, computer verification of their correctness is straightforward.

As a final example, consider the following specification for a sorting bag for integers.

SORTING BAG SPECIFICATION

Syntax:
ADD: integer
FRONT: → integer
REMOVE:

Semantics:
(1) L(T) → L(T.ADD(n).REMOVE)
(2) L(T) → T.ADD(n).FRONT ≡ T.ADD(n)
(3) V(T.ADD(n).T_1.FRONT)=n → T.ADD(n).T_1.REMOVE ≡ T.T_1
(4) V(ADD(n).FRONT)=n
(5) V(T.FRONT)<n → V(T.ADD(n).FRONT)=V(T.FRONT)
(5) V(T.FRONT)≥n → V(T.ADD(n).FRONT)=n

Intuitively, *ADD* adds integers to the bag, *FRONT* returns the smallest value of an integer in the bag, and *REMOVE* removes a smallest integer. The following program implements that bag:

SORTING BAG PROGRAM

```
ADD(n):
i:=1;
while (n<bag[i] & i≤tail) {i:=i+1};
bag:=INSERT(bag,i,n);
tail:=tail+1.

FRONT: RETURN(front)
front:=bag[tail].

REMOVE:
if (tail>0) then {tail:=tail-1}.
```

The implementation stores the inserted integers in descending order in an array and uses the procedure INSERT(array,index,integer), which inserts the given integer at the indexed location in the given array. For proving correctness, we can either use an implementation of INSERT or use a specification of INSERT if an implementation is not yet at hand. For definiteness, we assume that INSERT meets the following specification:

INSERT SPECIFICATION

Syntax:
INSERT: array × integer × integer → array

Semantics:
(1) V(T.INSERT(x,y,z))=w → w[y]=z
(2) V(T.INSERT(x,y,z))=w → (i<y → w[i]=x[i])
(3) V(T.INSERT(x,y,z))=w → (i>y → w[i]=x[i−1])

As before, to formulate correctness assertions we need program-relative versions of legality and equivalence. Notice that a trace is legal if it does not call *FRONT* or *REMOVE* on an empty bag, i.e., when *tail* =0. This gives us the following axiom for legality: $L(T) \longleftarrow (((\exists T_1)(T=T_1.FRONT \vee T=T_1.REMOVE) \wedge V(T,tail)>0) \vee V(T,tail)\geq 0)$. This axiom could be simplified if we altered our implementation to set *tail* to a negative integer when *FRONT* or *REMOVE* are illegally called. Then, the axiom $L(T) \longleftarrow V(T,tail)\geq 0$ would suffice. In general, program overkill can make program correctness easier to prove. As for equivalence, note that two traces are equivalent if they are co-legal and have the same bag. This leads us to the following axiom: $T_1 \equiv T_2 \longleftarrow ((L(T_1) \longleftarrow L(T_2)) \wedge V(T_1,tail)=V(T_2,tail) \wedge (1\leq i \leq V(T_1,tail) \rightarrow V(T_1,bag[i])=V(T_2,bag[i])))$. A correctness proof consists of a derivation of our trace axioms and program specification from our two new axioms, the bag implementation, and our program. As before we could use the two axioms to reformulate the trace axioms and program specification if desired.

5. CONCLUSIONS

We have found traditional methods for proving program correctness lacking in that they depend on unacceptable specification methods or require a leap of faith to bridge the gap between an acceptable specification and a program-relative one. In this report, we extended a formal abstract specification language to a program semantics language, compared the extended language with other program semantic languages, and showed how it can be used to prove program correctness *vis-a-vis* an acceptable specification.

6. REFERENCES

1. J. McLean, "Two Dogmas of Program Specification," *ACM SIGSOFT Eng. Notes* **10**, 85-87, Aug. 1985.

2. J. McLean, "A Formal Foundation for the Abstract Specification of Software," *J. ACM* **31**(3), 600-627 (1984).

3. W. Bartussek and D.L. Parnas, "Using Traces to Write Abstract Specifications for Software Modules," Report TR 77-012, University of North Carolina, Chapel Hill, NC, Dec. 1977.

4. J. McLean, "A Complete System of Temporal Logic for Specification Schemata," in *Logics of Programs*, D. Kozen, ed. (Springer-Verlag, New York, 1984), pp. 360-370.

5. R. Linger, H. Mills, and B. Witt, *Structured Programming: Theory and Practice* (Addison-Wesley, Reading, 1979).

6. W. Clocksin and C. Mellish, *Programming in Prolog* (Springer-Verlag, New York, 1981).

7. J. Lloyd, *Foundations of Logic Programming* (Springer-Verlag, New York, 1984).

8. E. Horowitz and S. Sahni, *Fundamentals of Data Structures* (Computer Science Press, Potomac, MD, 1977).

9. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM* **12**(10), 576-580 (1969).

10. E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, 1976).

11. D. Harel, *First-Order Dynamic Logic* (Springer-Verlag, New York, 1979).

12. J. McLean, "A Formal Foundation for the Trace Method of Software Specification," NRL Memorandum Report 4874, Sept. 1982.

7. ACKNOWLEDGMENTS

I am indebted to the participants of the foundation group at the 1985 VERKSHOP for convincing me that I should bite the bullet and abandon completeness to introduce induction into my program derivation system. Carl Landwehr provided useful comments on an earlier draft.

APPENDIX

This appendix contains derivations of the correctness assertions for our stack implementation given above. Proving assertions (1), (3), and (4) are easy given that a sequence of procedure calls or procedure call variables is a valid substitution instance of a statement variable. Assertion (4) follows directly from program semantic axiom (2), given the theorem for number theory that $V([],ptr)=0 \rightarrow V([],ptr)\geq 0$. Assertion (1) follows from number theory and our program semantic axioms given that $PUSH(n)=if\ (ptr\geq 0)\ then\ \{ptr:=ptr+1;stack[ptr]:=n\}$. By program axiom (9) $V(T,ptr)\geq 0 \rightarrow V(T.if\ (ptr\geq 0)\ then\ \{ptr:=ptr+1;stack[ptr]:=n\},ptr) = V(T.ptr:=ptr+1;stack[ptr]:=n,ptr)$, and $V(T.ptr:=ptr+1;stack[ptr]:=n,ptr) = V(T,ptr)+1$ by axioms (6), (3), (4) and (1). Hence, by number theory we have assertion (1). Assertion (3) follows by analogous reasoning given that $POP=if\ (ptr>0)\ then\ \{top:=stack[ptr]\}.ptr:=ptr-1$ and the fact that $V(T,ptr)\geq 0 \rightarrow V(T.PUSH(n),ptr)=V(T,ptr)+1$.

We prove assertion (2) in two parts, one for each disjunct of its consequent. $V(T,ptr)\geq 0 \rightarrow V(T,ptr)=V(T.PUSH(n).POP,ptr)$ is straightforward given our program semantic axioms and the implementations of $PUSH(n)$ and POP. This leaves $V(T,ptr)\geq 0 \rightarrow (1\leq i\leq V(T,ptr) \rightarrow V(T,stack[i])=V(T.PUSH(n).POP,stack[i]))$. Note that $i\leq V(T,ptr) \rightarrow V(T,i)\neq V\ (T.ptr:=ptr+1,ptr)$. Given this, deriving $V(T,ptr)\geq 0 \rightarrow (1\leq i\leq V(T,ptr) \rightarrow V(T,stack[i])=V\ (T.PUSH(n).POP,stack[i]))$ is straightforward using program semantic axioms (6) and (8). Assertion (2) follows by logic.

Instead of proving assertion (5) directly, we prove its contrapositive, $V(T,ptr)<0 \rightarrow V(T.T_1,ptr)<0$, by trace induction. Assertion (5) follows by simple sentential logic. Let P be the property such that $P(\phi)$ if and only if $V(\phi,ptr)<0 \rightarrow V(\phi.T_1,ptr)<0$. We prove $P(T)$, and thereby assertion (5), by using our induction schema. $P([])$ is $V([],ptr)<0 \rightarrow V([].T_1,ptr)<0$, which is trivial since $V([],ptr)=0$ by program axiom (2). $P(\phi) \rightarrow P(\phi.PUSH(i))$ is $(V(\phi,ptr)<0 \rightarrow V(\phi.T_1,ptr)<0) \rightarrow (V(\phi.PUSH(i),ptr)<0 \rightarrow V(\phi.PUSH(i).T_1,ptr)<0)$. We establish this by proving its consequent by induction. Let A be the property such that $A(\psi)$ if and only if $V(\phi.PUSH(i),ptr)<0 \rightarrow V(\phi.PUSH(i).\psi,ptr)<0$. We want to establish $A(T_1)$. $A([])$ is the assertion $V(\phi.PUSH(i),ptr)<0 \rightarrow V(\phi.PUSH(i).[],ptr)<0$, which is trivially true since $\phi.PUSH=\phi.PUSH(i).[]$. $A(\psi) \rightarrow A(\psi.PUSH(i))$ is the assertion $(V(\phi.PUSH(i),ptr)<0 \rightarrow V(\phi.PUSH(i).\psi,ptr)<0) \rightarrow (V(\phi.PUSH(i),ptr)<0 \rightarrow V(\phi.PUSH(i).\psi.PUSH(i),ptr)<0)$. Now $V(\phi.PUSH(i).\psi,ptr)<0 \rightarrow V(\phi.PUSH(i).\psi.PUSH(i),ptr)<0$, since $V(\phi.PUSH(i).\psi,ptr)<0 \rightarrow V(\phi.PUSH(i).\psi,ptr) = V(\phi.PUSH(i).\psi.PUSH(i),ptr)<0$. $A(\psi) \rightarrow A(\psi.PUSH(i))$ follows by sentential logic since $(B \rightarrow C) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$. $A(\psi) \rightarrow A(\psi.POP)$ is the assertion $(V(\phi.PUSH(i),ptr)<0 \rightarrow V(\phi.PUSH(i).\ \psi,ptr)<0) \rightarrow (V(\phi.PUSH(i),ptr)<0 \rightarrow V(\phi.PUSH(i).\ \psi.POP,ptr)<0)$. This holds by an analogous argument, establishing $A(T_1)$ and hence, $P(\phi) \rightarrow P(\phi.PUSH(i))$. $P(\phi) \rightarrow P(\phi.POP)$ holds by an analogous argument, which establishes $P(T)$, and we are done.

We prove assertion (6) by establishing four lemmas. First, consider (I) $V(T,ptr)=V(T_1,ptr) \rightarrow (V(T.T_2,ptr)\geq 0 \leftarrow V(T_1.T_2,ptr)\geq 0)$. Using induction on T_2, it is straightforward to establish that $V(T,ptr)=V(T_1,ptr) \rightarrow V(T.T_2,ptr)=V\ (T_1.T_2,ptr)$, from which (I) follows. Next consider (II) $(V(T,ptr)=V(T_1,ptr) \wedge (1\leq i\leq V(T,ptr) \rightarrow V(T,stack[i])=V\ (T_1,stack[i]))) \rightarrow (T_2\neq [] \rightarrow ((\exists v)V\ (T.T_2)=v \rightarrow V\ (T.T_2)=V\ (T_1.T_2)))$. Unabbreviating V, this translates into $(V(T,ptr)=V(T_1,ptr) \wedge (1\leq i\leq V(T,ptr) \rightarrow V(T,stack[i])=V(T_1,stack[i]))) \rightarrow (T_2.POP\neq [] \rightarrow ((\exists v)V(T.T_2.POP,top)=v \rightarrow V(T.T_2.POP,top)=V(T_1.T_2.POP,top)))$. Using induction on T_2, it is straightforward to prove $(V(T,ptr)=V\ (T_1,ptr) \wedge (1\leq i\leq V(T,ptr) \rightarrow V(T,stack[i])=V\ (T_1,stack[i]))) \rightarrow V(T.T_2.POP,top)=V\ (T_1.T_2.POP,top)$, from which (II) follows. Next, consider (III) $(T_2)\ (V(T.T_2,ptr)\geq 0 \leftarrow V$

$(T_1.T_2,ptr)\geq 0) \rightarrow V(T,ptr)=V\ (T_1,ptr)$. We establish (III) by proving $V(T,ptr)> V(T_1, ptr) \rightarrow (\exists T_2)\ (V(T.T_2,ptr)\geq 0 \rightarrow V(T_1.T_2,ptr)<0)$ from which (III) follows. By definition, $V(T,ptr)>V(T_1,ptr) \rightarrow (\exists x)(\exists y)(y>0 \wedge V(T_1,ptr)=x \wedge V(T,ptr)=x+y)$. Integer induction on x proves that $(x)(y)(y>0 \wedge x\geq 0 \wedge V(T_1,ptr)=x \wedge V(T,ptr)=x+y) \rightarrow (\exists T_2)(V(T.T_2,ptr)\geq 0 \wedge V(T_1.T_2,ptr)<0)$. For $x=0$, $T_2=POP$, and for $x=m+1$, $T_2=T_3.POP$ where T_3 works for m. Negative integer induction on x proves that $(y>0 \wedge x<0 \wedge V(T_1,ptr)=x \wedge V(T,ptr)=x+y) \rightarrow (\exists T_2)(V(T.T_2,ptr)\geq 0 \wedge V(T_1.T_2,ptr)<0)$. For $x=-1$, $T_2=PUSH(n)$, and for $x=m-1$, $T_2=T_3.PUSH(n)$, where T_3 works for m. This establishes (III). Finally, we prove (IV) $(T_2)((V(T.T_2,ptr)\geq 0 \longleftrightarrow V(T_1.T_2,ptr)\geq 0) \wedge (T_2\neq [] \rightarrow ((\exists v)V(T.T_2)=v \rightarrow V(T.T_2)=V(T_1.T_2)))) \rightarrow (1\leq i\leq V(T,ptr) \rightarrow V(T,stack[i])=V(T_1,stack[i]))$. Unabbreviating V, this translates to $(T_2)((V(T.T_2,ptr)\geq 0 \longleftrightarrow V(T_1.T_2,ptr)\geq 0) \wedge (T_2.POP\neq [] \rightarrow ((\exists v)\ V(T.T_2.\ POP,top)=v \rightarrow V\ (T.T_2.\ POP,top)=V\ (T_1.T_2.POP,top)))) \rightarrow (1\leq i\leq V(T,ptr) \rightarrow V(T,stack[i])=V\ (T_1,stack[i]))$. Using logic and lemma (III), (IV) follows from $(1\leq i\leq V(T,ptr) \wedge V(T,stack[i])\neq V(T_1,stack[i]) \wedge V(T,ptr)=V\ (T_1,ptr)) \rightarrow (\exists T_2)\ (T_2.POP\neq [] \wedge (\exists v)V\ (T.T_2.POP,top)=v \wedge V(T.T_2.\ POP,\ top)\neq V(T_1.T_2.\ POP,top))$. This last assertion can be proven by integer induction on i, and we are done.

END

6-87

DTIC